树的应用

最优二叉树

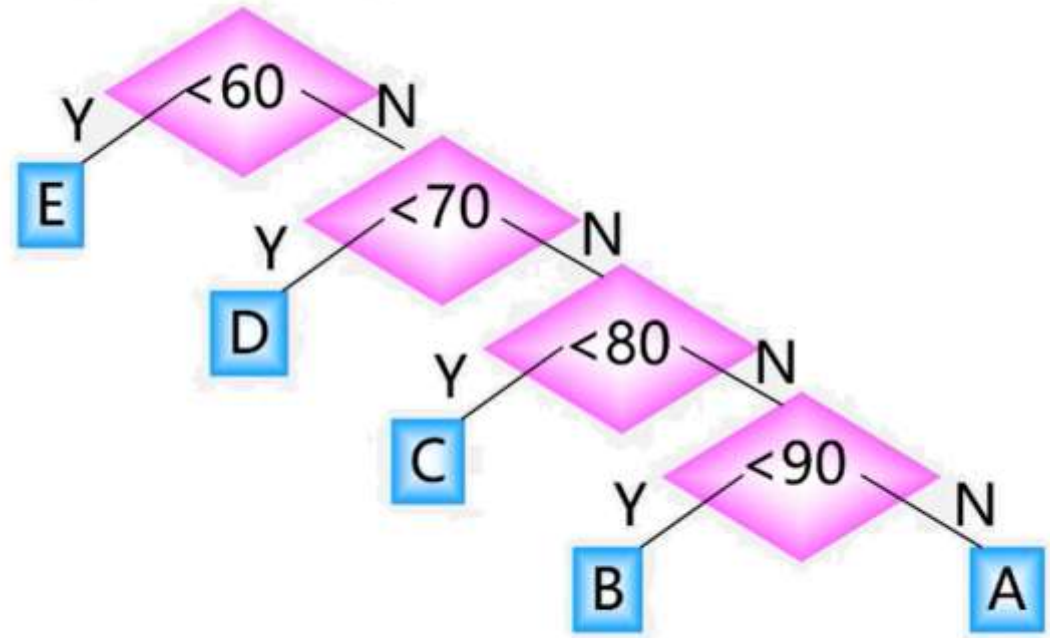
讲什么？

- 什么是最优二叉树（哈夫曼树）
- 哈夫曼算法
- 哈夫曼编码

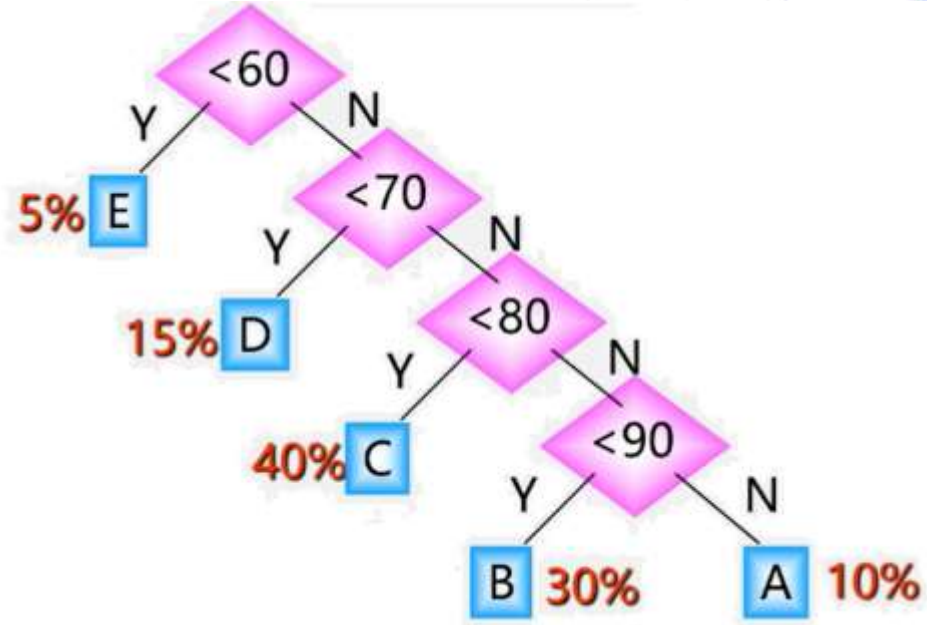
编程：将学生的百分制成绩转换为五分制成绩

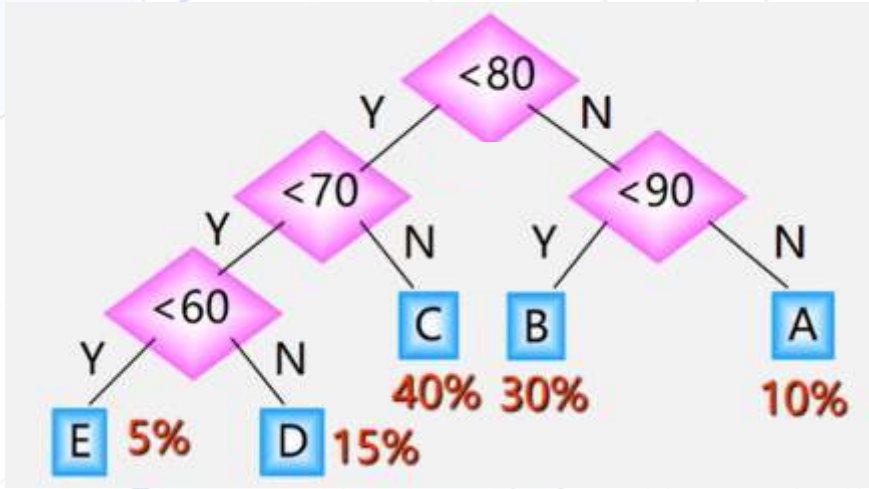
<60: E　　60-69: D　　70-79: C　　80-89: B　　90-100: A

```
if (score<60)
    grade== 'E' ;
else if (score<70)
        grade == 'D'
     else if (score<80)
            grade == 'C' ;
         else if (score<90)
                grade == 'B' ;
             else
                grade == 'A' ;
```



如果输入的数据量很大，需要考虑程序的操作时间。
例如，学生的成绩数据有10000个，预估5%不及格，15%在60-70之间，40%在70-80之间，30%在80-90之间，10%大于90.





左边这棵树对应的比较，5%的数据需比较1次，15%比2次，40%需比较3次，40%（30%+10%）需要比较4次，因此10000个数据比较的次数为10000*(1*5%+2*15%+3*40%+4*40%)=31500

右边这棵树对应的比较，需比较的次数为
10000*(2*30%+2*40%+2*10%+3*5%+3*15%)=22000

两种判断树的执行效率不同，如何找判断效率最高的树呢？

最优二叉树

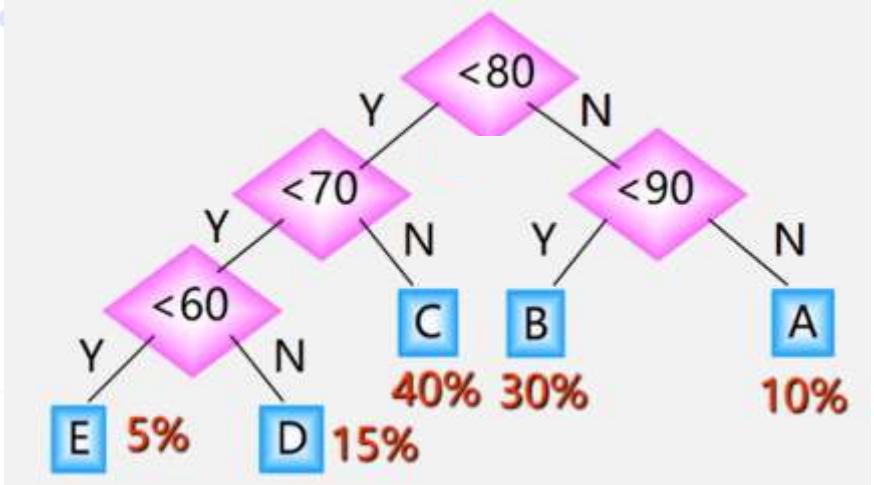
取决于具体问题

- 叶子结点的权值：对叶子结点赋予的一个有意义的数值量
- 二叉树的带权路径长度：从根结点到各个叶子结点的路径长度与相应叶子结点权值的乘积之和

$$\sum_{k=1}^{n} w_k l_k$$

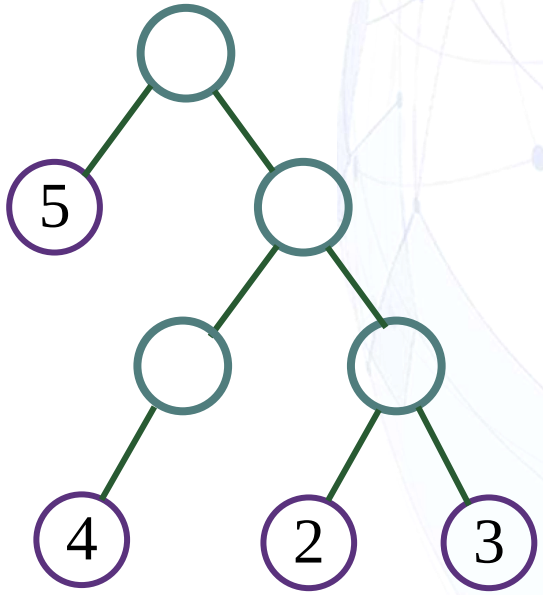
l_k：从根结点到第 k 个叶子的路径长度

w_k：第 k 个叶子的权值



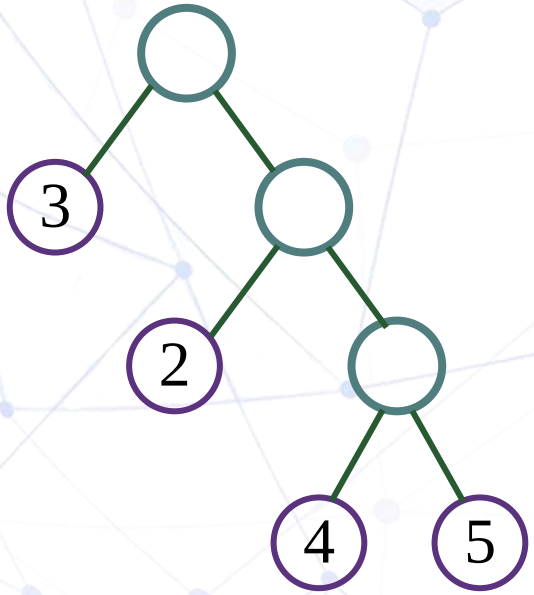
最优二叉树

- 最优二叉树（哈夫曼树, Huffman树）：给定一组具有确定权值的叶子结点，带权路径长度最小的二叉树



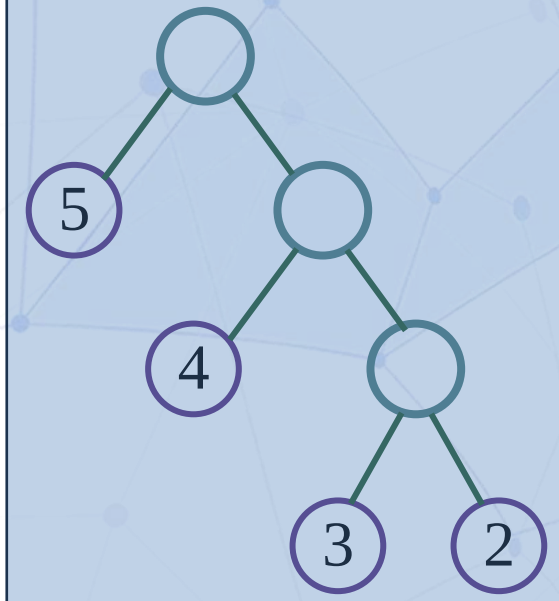
32

5*1+(4+2+3)*3=32



34

3*1+2*2+(4+5)*3=34



28

5*1+4*2+(3+2)*3=28

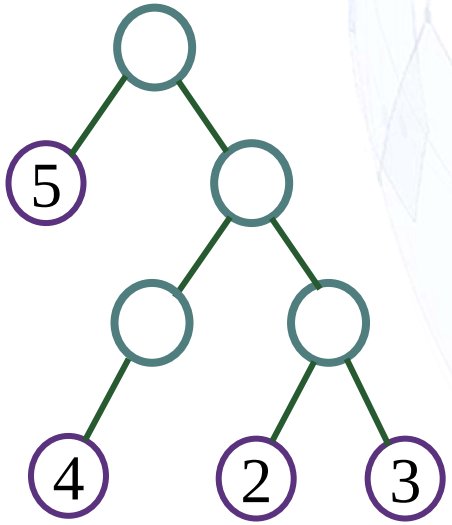
最优二叉树

最优二叉树有什么特点?

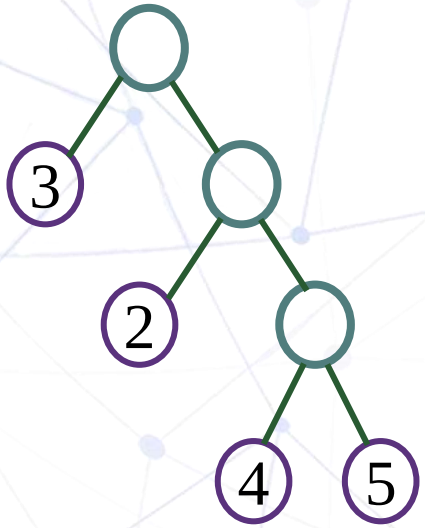
(1) 权值越大的叶子结点越靠近根结点

(2) 只有度为 0 和度为 2 的结点，不存在度为 1 的结点

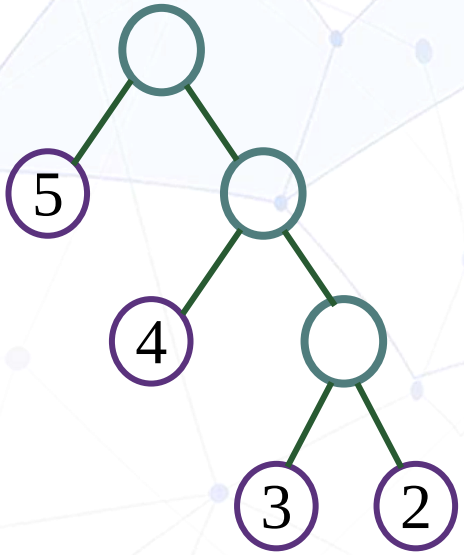
(3) 权重最小的和次小的叶结点处于最下层，且互为兄弟



5*1+(4+2+3)*3=32



3*1+2*2+(4+5)*3=34

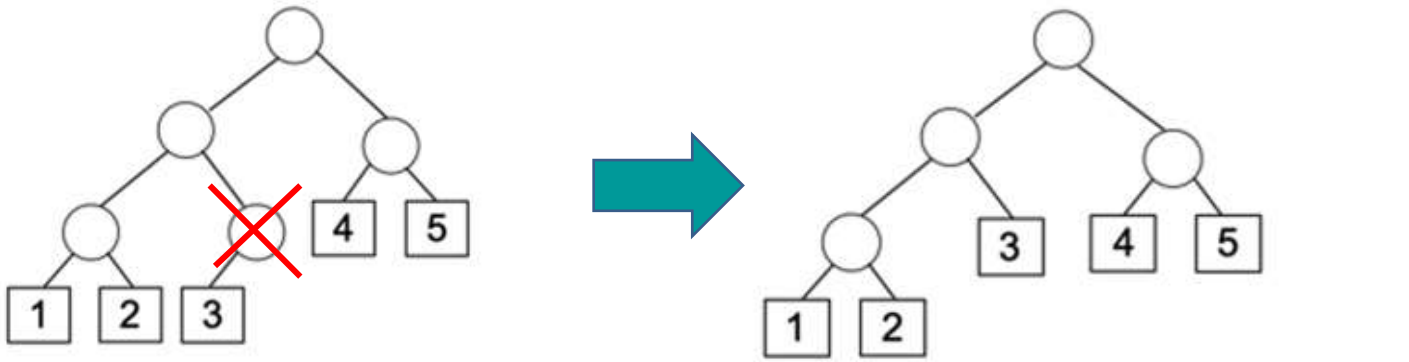


5*1+4*2+(3+2)*3=28

最优二叉树基本性质

定理5-4：最优二叉树（哈夫曼树）是满二叉树

证明：假设带权二叉树中存在度为1的中间结点。删除度为1的中间结点并把其唯一的子结点与父结点直接相连，使得从树根到该中间结点的所有子孙结点的路径长度减1，能够减小树的带权路径长度。因此，最优二叉树不含度为1的中间结点。



删除度为1的中间结点

减小树的带权路径长度，使其成为最优二叉树

最优二叉树基本性质

命题5-5：最优二叉树中，如果两个叶结点的权重值不同，则**权重值小**的叶结点在树中的**层数大于等于权重值大**的叶结点

证明：反证法。

- 设最优二叉树T的带权路径长度为WPL(T)，其中叶结点u的权重w_u小于叶结点v的权重w_v，即$w_u < w_v$
- 首先假定u在树中的层数比v的层数小，即level(u)<level(v)
- 交换叶结点u和v，可得新的带权二叉树T*, 且

$$WPL(T^*) = WPL(T) + w_u(level(v) - level(u)) + w_v(level(u) - level(v)) < WPL(T)$$

与T是最优二叉树矛盾。证明 level(u) ≥ level(v)。

交换权重值相同或者在树中同一层上的两个叶结点，不会改变二叉树的带权路径长度

最优二叉树基本性质

命题5-6：给定一组叶结点权重，存在最优二叉树，权重最小和次小的叶结点在树的最下层并且互为兄弟结点。

证明：

- 最优二叉树是满树，因此最下层必有两个以上的叶结点
- 如果权重最小的叶结点不在最下层，则最下层所有结点的权重都必须等于最小值，因此可以通过交换把权重最小的叶结点移到最优二叉树的最下层
- 同理可证权重次小的叶结点也在最下层
- 在最下层权重最小叶结点必有兄弟结点（满二叉树）。如果权重最小结点和次小结点不是兄弟，可以把最小结点的兄弟与次小结点交换，使两个结点共有一个父结点

【问题】给定一组权值，如何构造最优二叉树?

【想法——哈夫曼算法的基本思想】

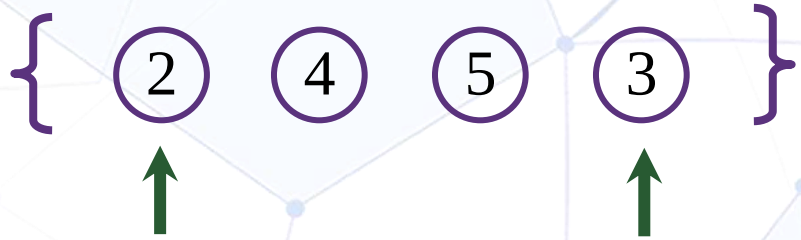
1. 初始化：由 n 个权值构造 n 棵只有一个根结点的二叉树，得到一个二叉树集合 $F=\{T_1, T_2, \ldots, T_n\}$;
2. 重复下述操作，直到集合 F 中只剩下一棵二叉树
   - 2.1 选取与合并：在 F 中选取根结点的权值最小的两棵二叉树分别作为左右子树构造一棵新的二叉树，这棵新二叉树的根结点的权值为其左右子树根结点的权值之和;
   - 2.2 删除与加入：在 F 中删除作为左右子树的两棵二叉树，并将新建立的二叉树加入到F中;

运行实例

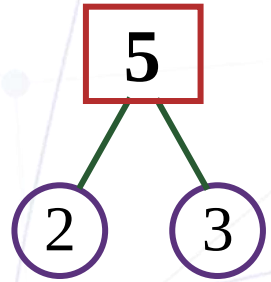
哈夫曼算法的运行实例

例 给定权值集合{2，4，5，3}

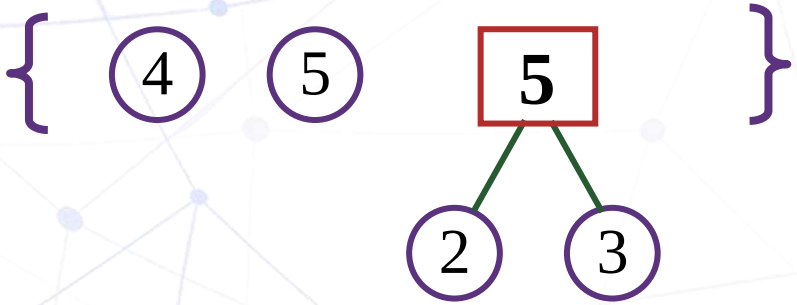
初始化



选取与合并



删除与加入



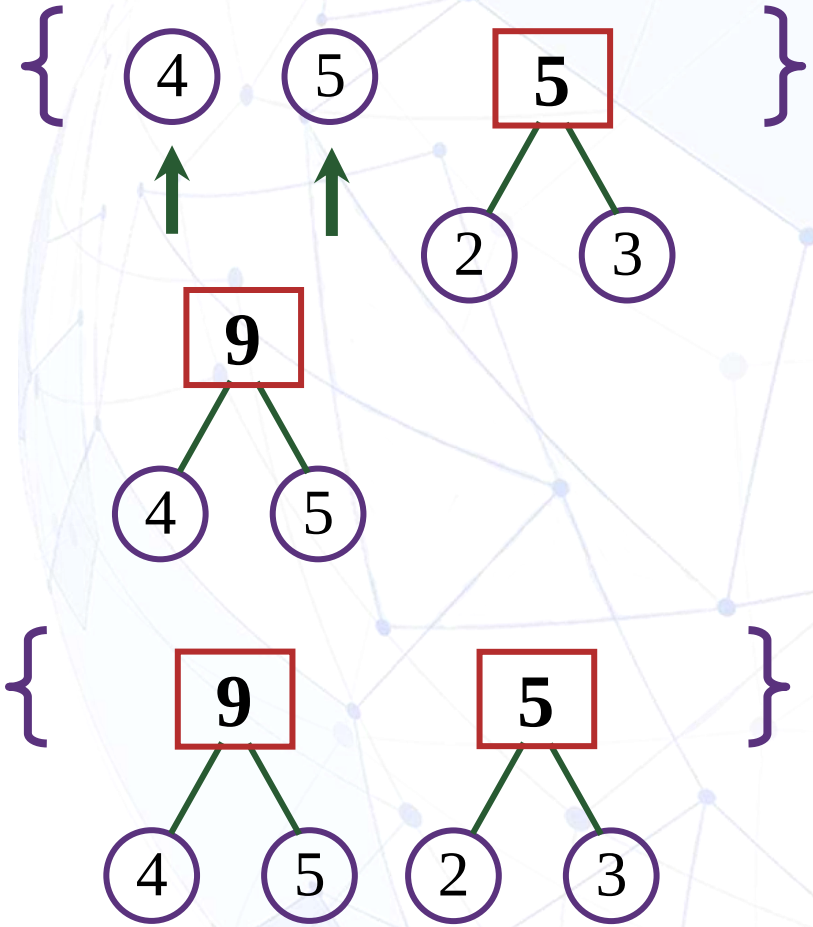
运行实例

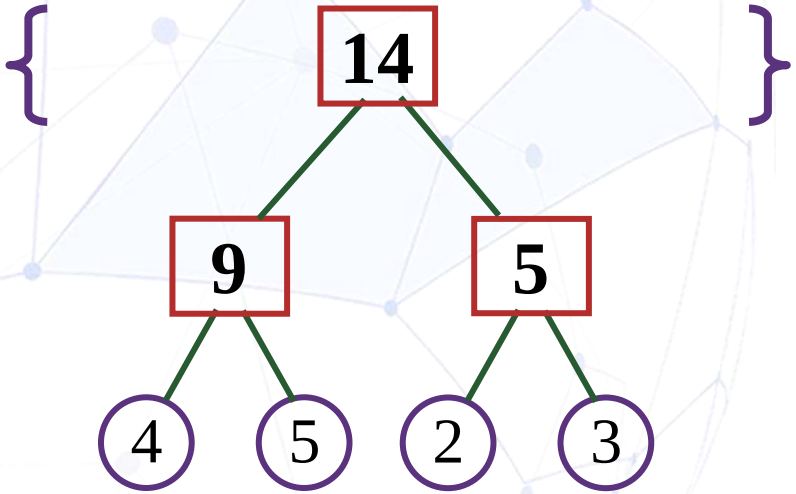
哈夫曼算法的运行实例

例 给定权值集合{2，4，5，3}

选取与合并

删除与加入





哈夫曼树不止一棵

哈夫曼算法

哈夫曼算法：一种至下而上构建最优二叉树的方法，通过不断合并两个带权二叉树，最终生成最优二叉树。

存储结构

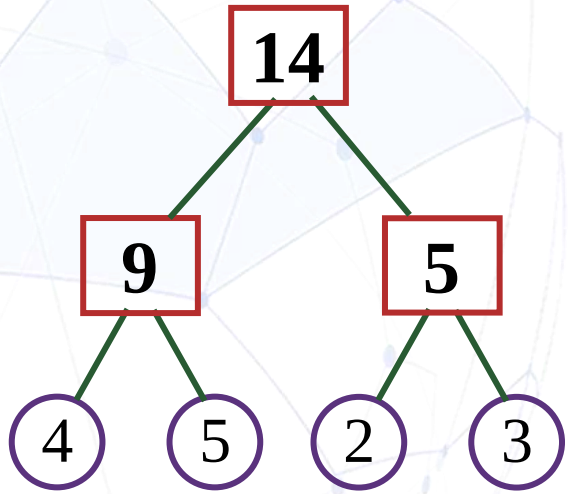
【算法——数据表示】如何存储哈夫曼树呢？

采用数组还是链表呢？

weight	lchild	rchild	parent

n 个叶子结点 ⇨ 合并 $n-1$ 次 ⇨ $n-1$个分支结点

哈夫曼树共有 $2n-1$个结点 ⇨ 设数组huffTree[2n−1]



须存储哪些关系呢？

选取根结点权值最小的二叉树 ⇨ 存储parent信息

作为左右子树进行合并 ⇨ 存储lchild和rchild信息

存储结构

weight	lchild	rchild	parent

```
struct ElemType
{
    int weight;
    int parent, lchild, rchild;
};
```

```
class HuffmanTree
{
public:
    HuffmanTree(int w[ ], int n);
    ~HuffmanTree( );
    void Print( );
private:
    ElemType *huffTree;
    int num;
    void Select(int n, int &i1, int &i2);
};
```

```
HuffmanTree :: HuffmanTree(int w[ ], int n)
{

}
```

算法描述

【算法——数据处理过程】

例 给定权值集合{2，4，5，3}

{ ② ④ ⑤ ③ }

算法描述：

```
for (i = 0; i < 2*n−1; i++)
{
    huffTree[i].parent = −1;
    huffTree[i].lchild = −1;
    huffTree[i].rchild = −1;
}
```

	weight	parent	lchild	rchild
0		-1	-1	-1
1		-1	-1	-1
2		-1	-1	-1
3		-1	-1	-1
4		-1	-1	-1
5		-1	-1	-1
6		-1	-1	-1

算法描述

【算法——数据处理过程】

例 给定权值集合{2，4，5，3}

{ ② ④ ⑤ ③ }

算法描述：

```
for (i = 0; i < n; i++)
    huffTree[i].weight = w[i];
```

	weight	parent	lchild	rchild
0	2	-1	-1	-1
1	4	-1	-1	-1
2	5	-1	-1	-1
3	3	-1	-1	-1
4		-1	-1	-1
5		-1	-1	-1
6		-1	-1	-1

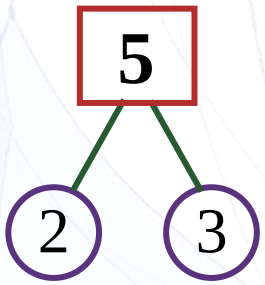
算法描述

【算法——数据处理过程】

例 给定权值集合{2，4，5，3}

{ 2 4 5 3 }

{ 4 5 5 }



	weight	parent	lchild	rchild
0	2	-1	-1	-1
1	4	-1	-1	-1
2	5	-1	-1	-1
3	3	-1	-1	-1
4	**5**	-1	-1	-1
5		-1	-1	-1
6		-1	-1	-1

i1 → 0

i2 → 3

k → 4

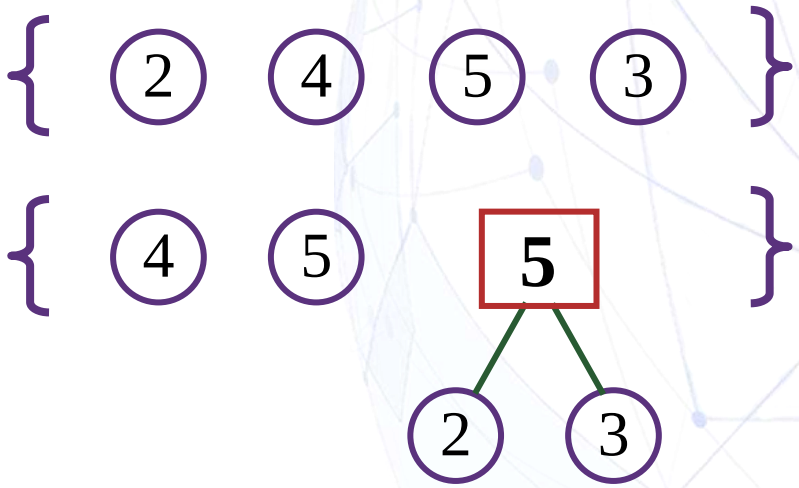
算法描述：

```
huffTree[k].weight = huffTree[i1].weight + huffTree[i2].weight;
```

算法描述

【算法——数据处理过程】

例 给定权值集合{2，4，5，3}



i1 → 0

i2 → 3

k → 4

	weight	parent	lchild	rchild
0	2	4 ~~-1~~	-1	-1
1	4	-1	-1	-1
2	5	-1	-1	-1
3	3	4 ~~-1~~	-1	-1
4	5	-1	0 ~~-1~~	3 ~~-1~~
5		-1	-1	-1
6		-1	-1	-1

算法描述：

```
huffTree[i1].parent = k;  huffTree[i2].parent = k;
huffTree[k].lchild = i1;  huffTree[k].rchild = i2;
```

算法描述

```
HuffmanTree :: HuffmanTree(int w[ ], int n)
{
    int i, k, i1, i2;
    huffTree = new ElemType [2*n-1];
    num = n;
    for (i = 0; i < 2*num-1; i++)
    {
        huffTree[i].parent = -1;  huffTree[i].lchild = huffTree[i].rchild = -1;
    }
    for (i = 0; i < num; i++)
        huffTree[i].weight = w[i];
    for (k = num; k < 2*num-1; k++)
    {
        Select(k, i1, i2);     //找到权值最小的根结点下标:i1和i2,具体实现略;
        huffTree[k].weight = huffTree[i1].weight + huffTree[i2].weight;
        huffTree[i1].parent = k; huffTree[i2].parent = k;
        huffTree[k].lchild = i1; huffTree[k].rchild = i2;
    }
}
```

算法5-15：创建哈夫曼树 CreateHuffmanTree(*w*)

输入：权重值的数据集*w*，$|w| \geq 2$
输出：哈夫曼树

tree_set ← ∅ //二叉树集合的初始化
n ← Length(*w*) //*n*个权重
for *i* ← 1 **to** *n* **do** //初始化*n*个二叉树
| *tree* ← **new** BinaryTreeNode( ) //创建叶结点
| *tree.left* ← NIL
| *tree.right* ← NIL
| *tree.weight* ← Extract(*w*) //从数据集*w*中取出一个值，作为结点权重
| Insert(*tree_set*, *tree*) //将单结点二叉树放入集合tree_set
end
| //合并二叉树

- 时间$T_{W_Extract}$
- 时间T_{Q_Insert}

算法5-15：创建哈夫曼树 CreateHuffmanTree(*w*)

```
|
for i ← 1 to n-1 do      //合并二叉树，共n-1次
| tree ← new BinaryTreeNode( )      //新建树根结点
| tree.left ← ExtractMin(tree_set)      //取出根权重最小树作为左子树
| tree.right ← ExtractMin(tree_set)      //取出根权重次小树作为右子树
| tree.weight ← tree.left.weight + tree.right.weight  //设置新树的根权重
| Insert(tree_set, tree)                              //将新树插入集合tree_set
end
tree ← Extract(tree_set)      //取出集合中唯一的二叉树，即哈夫曼树
return tree
```

- 时间$T_{Q_ExtractMin}$
- 时间$T_{Q_ExtractMin}$
- 时间T_{Q_Insert}
- 时间$T_{Q_Extract}$

哈夫曼算法

哈夫曼算法：一种至下而上构建最优二叉树的方法，通过不断合并两个带权二叉树，最终生成最优二叉树

算法分析：

- 用n个权重值创建了n个单根二叉树，并依次放入集合tree_set中，时间复杂度为$O(n)*(T_{Q_Insert} + T_{W_Extract})$
- 合并两个二叉树的时间是$2T_{Q_ExtractMin} + T_{Q_Insert}$
- 算法的时间复杂度等于 $O(n)*T_{Q\_Insert} + O(n)*T_{Q\_ExtractMin} + O(n)*T_{W\_Extract}$
- 假设数据集w和tree_set直接用线性表实现，T_{Q_Insert}和$T_{W_Extract}$都是$O(1)$，但$T_{Q_ExtractMin} = O(n)$，即在tree_set中顺序查找权重最小二叉树的时间，因此构建哈夫曼树的时间复杂度达到$O(n^2)$
- 更高效的构建方案是使用比线性表更复杂的数据结构，比如堆，这将在第6章中介绍。

单选题 5分

设置

1. 最优二叉树中不存在度为 1 的结点。

A 正确

B 错误

提交

单选题 5分

设置

2. 在最优二叉树中，权值越大的叶子结点越靠近根结点。

A 正确

B 错误

提交

单选题 5分

设置

3. 在最优二叉树中，根结点的值等于所有叶子结点的权值之和。

A 正确

B 错误

提交

单选题 5分

设置

4. 给定权值{3, 4, 5, 6, 7}，采用哈夫曼算法构造最优二叉树，带权路经长度是（ ）。

A 25

B 57

C 62

D 61

提交

主观题 10分

设置

5. 哈夫曼算法如何存储哈夫曼树？为什么？

正常使用主观题需2.0以上版本雨课堂

作答

编码与解码

- 编码：给每一个对象标记一个二进制位串来表示一组对象
- 等长编码：用长度相等的二进制位串表示一组对象

酸	甜
0	1

酸	甜	苦	辣
00	01	10	11

酸	甜	苦	辣	咸	麻
000	001	010	011	100	101

	省	市	区	出生日期	顺序	校验
身份证号码	22	01	04	19870126	444	7

- 等长编码的优势：编码和解码操作简单

比如ASCII码（1个字节），Unicode编码（2个字节）等

- 编码的目的是什么?

  ⇨ 数字化 ⇨ 编码效率取决于编码长度

- 如果每个对象的使用频率不等，如何获得较高的编码效率?

编码与解码

- 如果每个对象的使用频率不等，如何获得较高的编码效率?

baaacabwbzc

a出现4次，b出现3次，c出现2次，w和z出现1次

a和b出现次数最多，如果缩短它的编码长度，而相应增加W和Z的码长，可以缩短编码长度

- 不等长编码的主要思路：让出现频率高的字母采用短编码，而频率低的采用长编码

编码与解码

不等长编码：表示一组对象的二进制位串的长度不相等

设计不等长编码时，必须考虑解码的唯一性

一组对象	A	B	C	D	E
使用频率	35	25	15	15	10
不等长编码	0	1	01	10	11

AABACD　　00100110

0 0 100110

0 01 00110

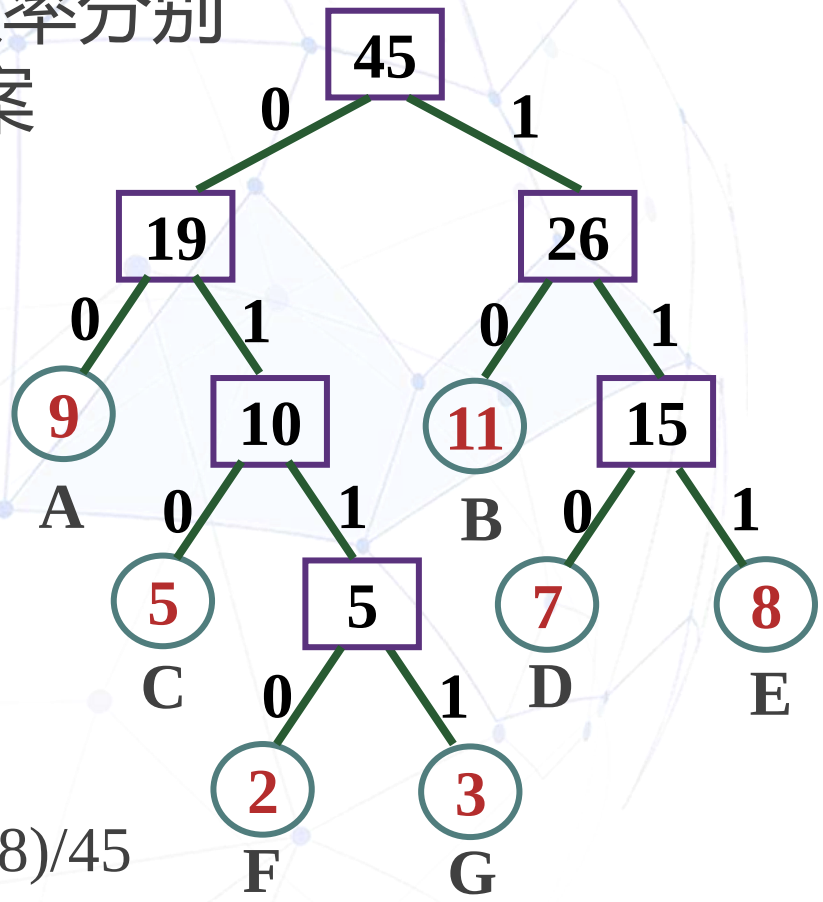
前缀编码：在一组编码中，任一编码都不是其它任何编码的前缀

前缀（无歧义）编码保证了在解码时不会有多种可能

哈夫曼编码

如何设计一个编码效率最高的前缀编码呢？⇨ 哈夫曼编码

例：一组字符{A, B, C, D, E, F, G}出现的频率分别是{9, 11, 5, 7, 8, 2, 3}，设计最经济的编码方案



哈夫曼编码方案：

A：00

B：10

C：010

D：110

E：111

F：0110

G：0111

等长编码的长度：3

哈夫曼编码的平均长度：

$(2\times9+3\times5+4\times2+4\times3+2\times11+3\times7+3\times8)/45 = 2.67$

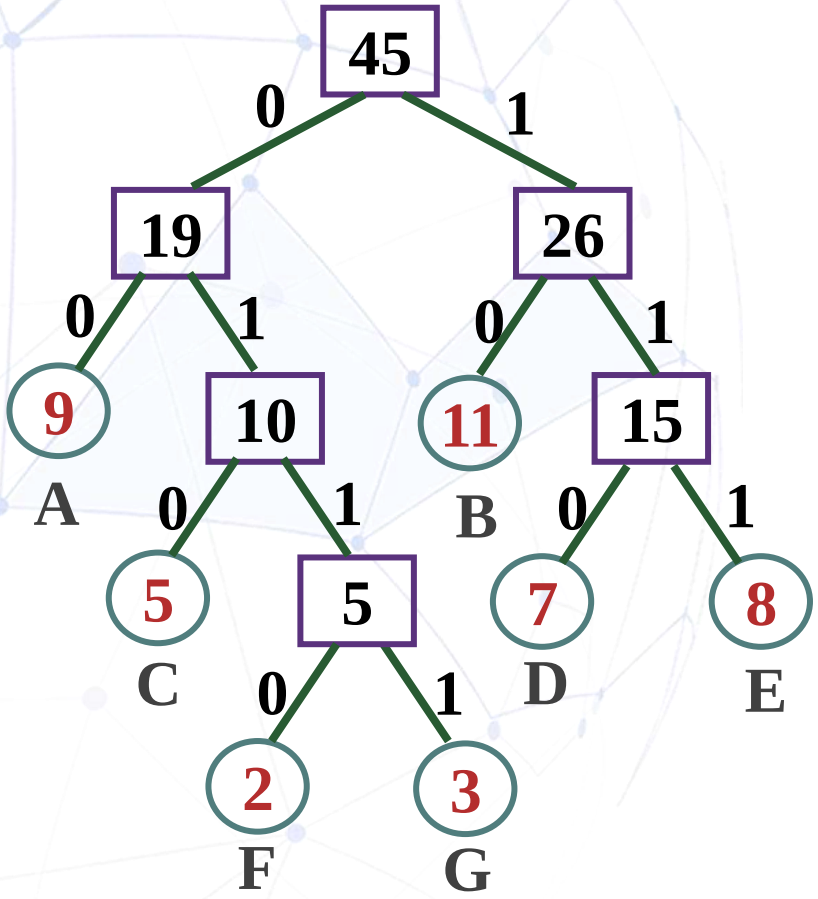
哈夫曼编码

对“BACBAD”进行编码：

10000101000110

哈夫曼编码方案：

A：00
B：10
C：010
D：110
E：111
F：0110
G：0111



对“10000101000110”进行解码：

10B-00A-010C-10B-00A-110D

哈夫曼解码算法

算法5-16：使用哈夫曼树对二进制字符串解码 Decoding(*tree*, *binary_code*)

输入：非空前缀码树*tree*，二进制字符串*binary_code*
输出：解码后的文字符序列

```
p ← tree    //指向树根
n ← Length(binary_code)  //二进制字符串长度
for i ← 0 to n-1 do
| if binary_code[i] = 0 then
| | p ← p.left      //遇到0, 沿左分支下移
| else   //binary_code[i] = 1
| | p ← p.right     //遇到1, 沿右分支下移
| end
| if p.left = NIL 且 p.right = NIL then  //到达叶结点
| | print  p.data     //输出文字符
| | p ← tree         //返回树根，重新开始解码
| end
end
```

算法分析

对长度为n的二进制字符串，使用哈夫曼（前缀码）树只需要O(n) 的时间就能解码生成原来的文字串

单选题 5分

设置

6. 通常来说，不等长编码的编码效率高于等长编码的编码效率。

A 正确

B 错误

提交

单选题 5分

设置

7. 在一组编码中，如果某个编码是其他编码的前缀，则解码可能不唯一。

A 正确

B 错误

提交